Roborodentia Robot
Senior Project, Spring 2015

Authors: Jordan Dykstra, Anibal Hernandez, Robert Prosser
Advisor: Dr. John Seng

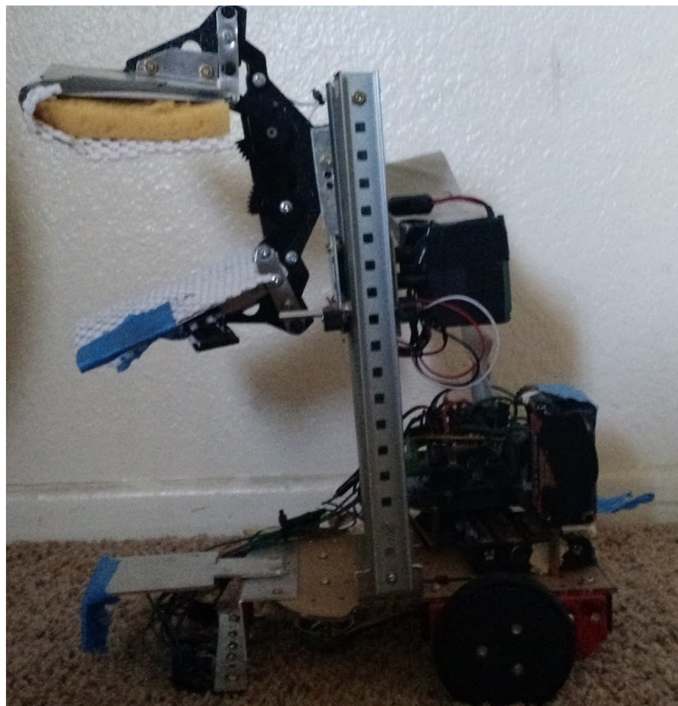


Table of Contents

Introduction 2
Problem Statement 2
Software 3
- Input Data 3
- The State Machine 5
- Wheel Drivers 9
- Rack Motor Driver 11
- Claw PWM 11
- Misc Information About Code 13

Hardware 13
- Line Sensors 14
- Beam Break Sensor 14
- Rack Motor 15
- Servos 16
- Power Circuit 17

Mechanical Design 17
- Claw 17
- Rack and Pinion 20
- Front Bumper/Line Sensor Mount 22

Budget and Bill of Materials 24
Lessons Learned 26
- Mechanical Aspects of Parts and Their Integration 26
- Using appropriate wires and heat shrink 26
- Pay more attention to power usage 26
- Line sensors aren't great for detecting wheel revolutions 26
- Don't use claw 27
- Use better motors 27
- Bumper button 27
- Designed for simplicity for reliability, maybe too simple 27

Conclusion 28
Appendix A: Code 29

Introduction

Roborodentia is an annual robotics competition at Cal Poly. Robots complete a task to acquire points. For the 2015 competition, robots moved rings from one side of a course to the other (of course it's a bit more complex than that, as will be discussed in the problem statement section). Each match was 1 on 1, and the robot who acquired the most points would win that match. A double elimination bracket was used to determine the winners. Also, the robots were completely autonomous (no external control). The team designed a robot for the competition and named him Steve.

Problem Statement

As mentioned previously, the goal is to move rings from one side of the course to the other. See the course in **Figure 1** below.

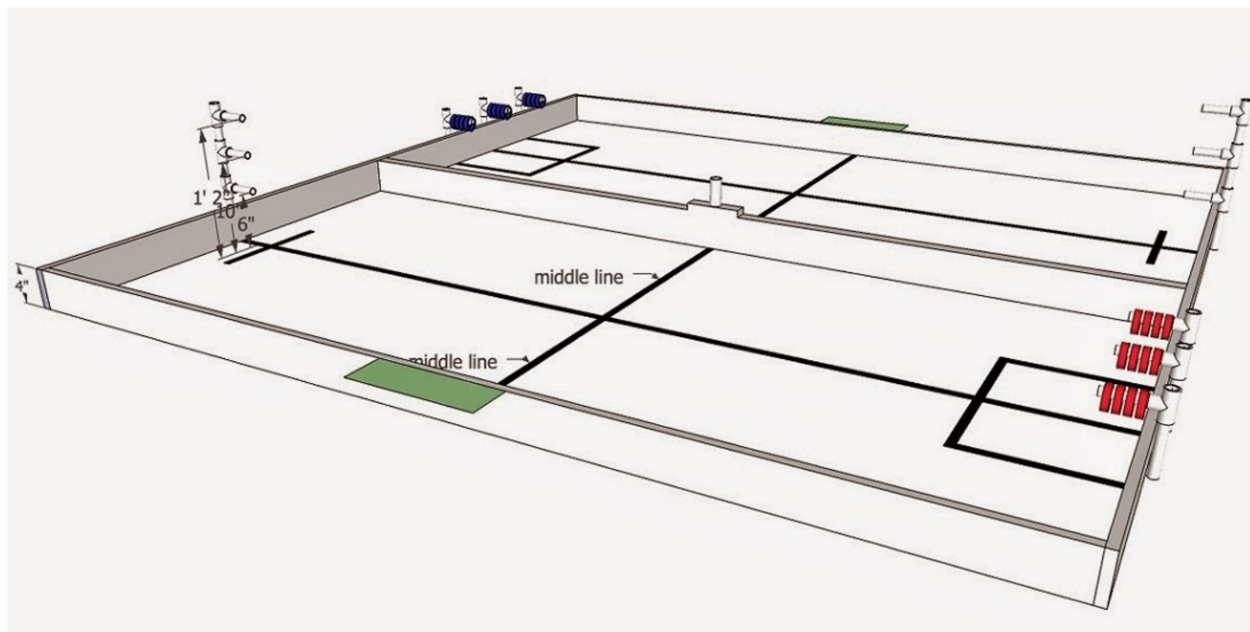


Figure 1: The Roborodentia 2015 course

There are several rules that make the design of the robot more challenging. The robot has to be less than twelve inches in height at the start of the match. Notice, however, that the highest vertical peg is fourteen inches high. This means that the robot must extend up somehow if it wants to reach that peg. There are width/length requirements as well: eleven inches by eleven inches at the start which can expand to a max of thirteen by thirteen inches during the match.

Each vertical peg has a different scoring value. Rings placed on the lowest peg are worth two points, the middle four points, and the top six points. If a robot has rings on all scoring pegs simultaneously, a fourteen point bonus is awarded and the rings are removed. Otherwise, rings are removed once four are on a vertical peg and the robot crosses the middle line.

Also note the vertical peg in the middle of the course. The robot with the highest ring on the middle peg at the end of the match will have its score doubled.

Finally, a robot can be reset during the match. However, the first reset will result in three points being given to the other team, the second four points, etc.

Only the most relevant rules have been listed here since the list is rather long. The complete rule list is listed here:
https://docs.google.com/document/d/1hEpUtLgn5UAsiko7OFaVtgVSniHXmVvJ0WFP1OPBKhI/pub

Software

The software was written in AVR C for the Arduino Uno microcontroller.

The block diagram for the system is shown in **Figure 2** below.

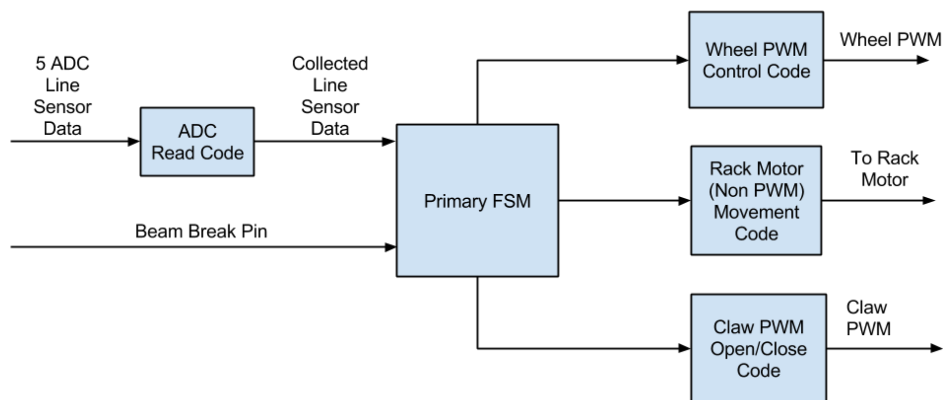


Figure 2: High level software block diagram for the system

Input Data

The system collects input data from five analog line sensors and a digital beam break sensor. The line sensor data is used as one of the primary drivers for the state machine and helps the robot to determine it's position. The beam break sensor is used to determine when the claw has

moved up or down to the desired position (for a more detailed explanation of how the beam break is used to determine the claw's position, see the mechanical design section of the report).

All five analog sensors output analog data. In order for the arduino to interpret this data, it must be driven through the ADC. The Arduino Uno ADC can only process one ADC input channel at a time. This means that the data from each channel must be read sequentially, as demonstrated in the ADC read function shown in **Figure 3** below.

```
void AvgReadAllADCValues(adcData *data)
{
   uint16_t ind = 0;
   uint16_t samples = 0;
   uint16_t max_sample = 2;

   for (ind = 0; ind < data->len; ++ind)
      data->readings[ind] = 0;

   while (samples < max_sample)
   {
      ind = 0;

      //read all desired adc values into the adcData struct
      while (ind < data->len)
      {
         //Select the current adc mux value
         ADMUX &= ~(0xF); //clear lower 4 bits, which are the mux values
         ADMUX |= ind; //set the correct mux value
         ADCSRA |= (1 << ADSC); //Start measuring!
         while (ADCSRA & (1 << ADSC))  //wait for measurement to complete
            ;
         data->readings[ind] += ADCH; //add the reading to the data
         ++ind;
      }

      ++samples;
   }

   for (ind = 0; ind < data->len; ++ind)
      data->readings[ind] /= samples;
}
```

Figure 3: Function to read ADC data

Note how the data is read sequentially off of each analog input pin based on the ADMUX value (an average is collected to help acquire more reliable input data). All of the data is read into an adcData struct, as shown in **Figure 4** on the next page.

```
typedef struct adcData {
   uint16_t len;
   int16_t readings[NUM_ADC];
} adcData;
```

Figure 4: struct that contains adc data

The second piece of input is from the beam break sensor, as mentioned earlier. The beam break input is used by the MoveClawUp and MoveClawDown functions, an example of which is shown in **Figure 5** below.

```
void MoveClawDown()
{
   PORTD |= RACK_MOTOR_RED;
   PORTD &= ~(RACK_MOTOR_BLACK);
   _delay_ms(500);
   while (PINB & BEAM_BREAK_INPUT)
      ;
   PORTD &= ~(RACK_MOTOR_RED);
   PORTD &= ~(RACK_MOTOR_BLACK);
}
```

Figure 5: MoveClawDown code that utilizes the beam break input

The function drives the rack motor's red wire high and the rack motor's black wire low (this is done by enabling pins on the motor control chip, which will be discussed later in the report). This will result in the motor moving the claw down. There is a short delay because the beam break will always start at a location where the beam is being broken. The delay allows the beam break sensor to move past this position. The beam break input pin is then polled until it goes low (when the beam is broken, the output is low). This means the claw has reached the desired destination. The wires are then both driven low to stop the motor.

The State Machine

The state machine code is the heart of the system. It determines what the robot should be doing at every moment in time.

The state machine begins by reading data from the line sensors. Some logic (shown in **Figure 6** on the next page) is then run which is specific to state 0 and state 6. State 0 and 6 are the states in which the robot follows a black line until horizontal lines have been crossed. These states correspond to when the robot moves from one end of the board to the other.

```
AvgReadAllADCValues(&data);
 if ((data.readings[OUTER_LEFT_LINE_SENSOR] > THRESHOLD ||
       data.readings[OUTER_RIGHT_LINE_SENSOR] > THRESHOLD) &&
        (state == 0 || state == 6) && blackOn == 0)
 {
    StopLeftWheel();
    StopRightWheel();
    ++crossCount;
    blackOn = 1;
 }
 else if ((data.readings[OUTER_LEFT_LINE_SENSOR] < THRESHOLD &&
    data.readings[OUTER_RIGHT_LINE_SENSOR] < THRESHOLD)&& blackOn == 1)
       blackOn = 0;

 if (crossCount == 2)
 {
    ++state;
    ++crossCount;
 }
```

Figure 6: Initial State Machine Code

In both states 0 and 6, the robot must detect two horizontal lines (represented by the crossCount variable). If an outer left line sensor is activated, this means that a horizontal line has been crossed. The check isn't run again until the robot has passed the horizontal line.

Once the crossCount variable has reached two, the state progresses. The code in the actual states merely performs line following (refer to **Figure 7** on the next page).

```
case 0: //follow the line
     if (data.readings[MIDDLE_LINE_SENSOR] > THRESHOLD)
     {
        LeftWheelForward();
        RightWheelForward();
     }
     else if (data.readings[INNER_LEFT_LINE_SENSOR] > THRESHOLD)
     {
        StopLeftWheel();
        RightWheelForward();
        _delay_ms(MOVE_DELAY);
     }
     else if (data.readings[INNER_RIGHT_LINE_SENSOR] > THRESHOLD)
     {
        StopRightWheel();
        LeftWheelForward();
        _delay_ms(MOVE_DELAY);
     }

     break;
```

Figure 7: Case 0 and Case 6 code

The code follows the line by interpreting data from the inner left, middle, and inner right line sensors. The inner left and inner right line sensors should not be detecting the line (only the middle line sensor should be on the line). If a line sensor detects it is on the line, the wheel corresponding to that sensor is stopped for a small amount of time to allow the robot's trajectory to recover. For example, if the inner left line sensor detects it is on the line (meaning the robot is veering to the right), then the code stops the left wheel for a moment and the robot turns left slightly to correct.

After the robot has progressed past state 0, it needs to open the claw and move forward. State 1 sets up a timer that has the robot go forward for an empirically determined amount of time until it hits the wall (a bumper prevents damage). State 2 line follows until the timer goes off. Once the timer goes off, an interrupt occurs that puts the robot into state 3.

In state 3, the robot's claw closes around the rings. The code then immediately jumps to state 4. In state four, the robot backs up until it hits a horizontal line (using essentially the same line following code as state 0 and state 6). After the horizontal line is hit, the state changes to state 5. Refer to **Figure 8** on the next page for the code.

```
case 5:
    //turn right until middle line sensor hits middle line
    LeftWheelReverse();
    RightWheelForward();
    _delay_ms(500);

    AvgReadAllADCValues(&data);
    while (data.readings[MIDDLE_LINE_SENSOR] < THRESHOLD)
       AvgReadAllADCValues(&data);
    StopLeftWheel();
    StopRightWheel();
    turning = 0;

    state = 6;
    crossCount = 0;
    blackOn = 0;
    break;
```

Figure 8: Turn around code

The left wheel is put into reverse mode and the right wheel is put into forward mode. This results in the robot turning left. There is a slight delay before the line detection code begins so that the middle line sensor can get past the horizontal line. Data is then repeatedly read until the middle line sensor hits a black line. CrossCount is then set to 0 to prepare for state 6. State 6 repeats the same process of line following until two lines are crosses exactly like in state 0.

After the second horizontal line is hit, the state changes to state 7. State 7 sets up an interrupt for state 8 which when triggered will change the state to 9. State 8 reverse line follows until the interrupt is activated. The reverse line following is necessary because when the robot hits the horizontal line, the claw/rings are actually on the lowest peg. The robot therefore needs to back up so that the claw can be raised.

In state 9, the robot stops moving then moves the claw up. The robot then changes to state 10 and line follows until the second horizontal line is hit again. The state then changes to 11.

In state 11 the timer/interrupts are setup so that the robot will go forward for a specific amount of time in state 12. After the timer expires, the state updates to state 13. In state 13 the claw opens and the rings are released onto the pegs. A timer is then setup for reverse line following and the state is updated to 14.

In state 14 the robot reverse line follows until the timer expires and the state is then updated to state 15. The robot then turns around like in state 5. The robot then stops and the claw is moved down. The state machine is then reset to allow the entire process to repeat and more points to be collected! See **Figure 9** on the next page for a pictorial representation of the whole state machine process.

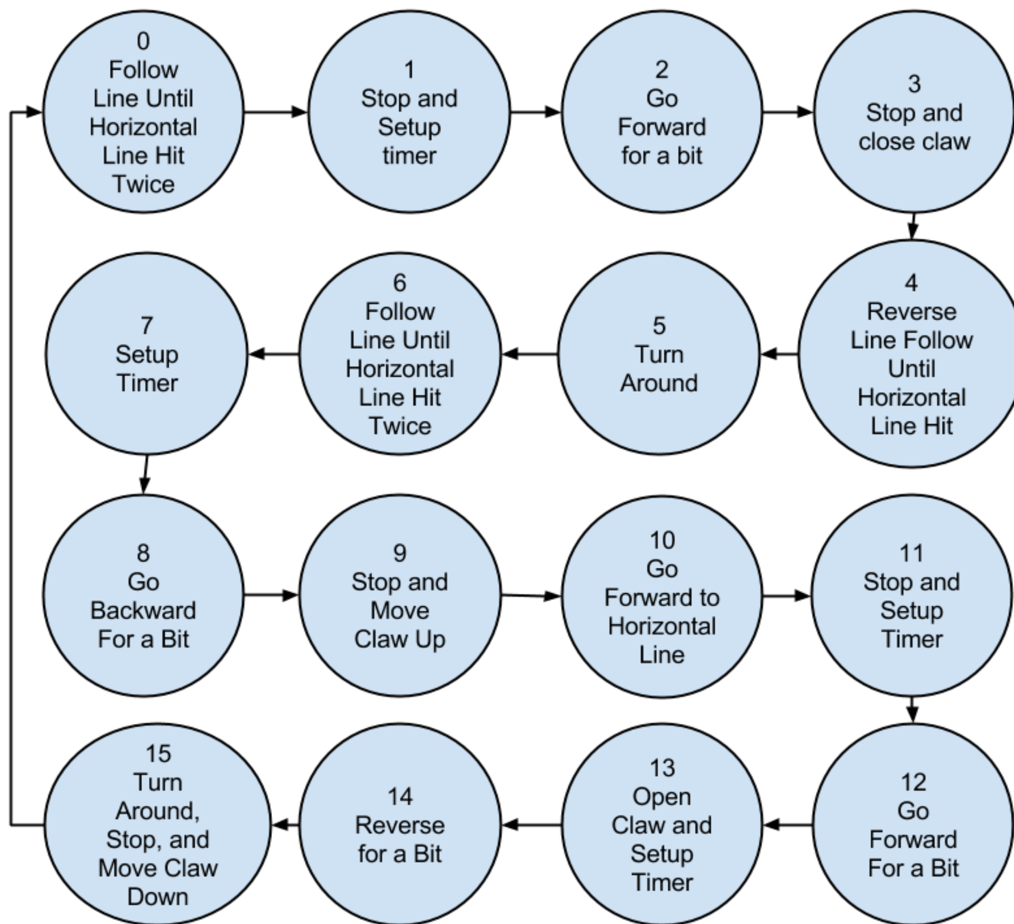


Figure 9: The complete state diagram for the system

Wheel Drivers

The wheels are controlled by pulse width modulated square waves. The waves run at 50 HZ (a 20 ms period). Depending on the duty cycle, the wheels move forward, backward, or are stopped. A 2 ms high pulse corresponds to moving a wheel forward, and a 1 ms high pulse corresponds to moving a wheel in reverse. Refer to **Figure 10** on the next page for the interrupt code.

```
ISR(TIMER2_COMPA_vect) {
   switch (motorState)
   {  //16 MHz, 1024 prescaler -> 64 uS periood
      case 0:
         OCR2A = 255;
         TCNT2 = 0;
         ++motorState;
         break;
      case 1:
         OCR2A = 5;
         TCNT2 = 0;
         ++motorState;
         break;
      case 2: //2 ms pulse
         if (leftWheelState == LEFT_WHEEL_FORWARD)
            PORTD |= LEFT_WHEEL_PIN;
         if (rightWheelState == RIGHT_WHEEL_FORWARD)
            PORTD |= RIGHT_WHEEL_PIN;
         if (leftWheelState == LEFT_WHEEL_STOP)
            PORTD &= ~(LEFT_WHEEL_STOP);
         if (rightWheelState == RIGHT_WHEEL_STOP)
            PORTD &= ~(RIGHT_WHEEL_STOP);
         OCR2A = 8;
         TCNT2 = 0;
         ++motorState;
         break;
      case 3: //1.5 ms pulse
         OCR2A = 8;
         TCNT2 = 0;
         ++motorState;
         break;
      case 4: //1 ms pulse
         if (leftWheelState == LEFT_WHEEL_REVERSE)
            PORTD |= LEFT_WHEEL_PIN;
         if (rightWheelState == RIGHT_WHEEL_REVERSE)
            PORTD |= RIGHT_WHEEL_PIN;
         OCR2A = 16;
         TCNT2 = 0;
         ++motorState;
         break;
      case 5:
         PORTD &= ~(LEFT_WHEEL_PIN | RIGHT_WHEEL_PIN);
         motorState = 0;
         break;
   }
}
```

Figure 10: Motor driving ISR Code

The code uses a 8 bit timer. OCR2A represents how many timer ticks will occur before generating an interrupt. The timer has a 1024 prescaler for the 16 MHz clock, which means that the timer will tick every 64us. This isn't exact in practice (most likely because of interrupt overhead), so the value of OCR2A to result in 18ms was determined with an oscilloscope.

The states orchestrate the interrupts so that one occurs at 18ms, 18.5 ms, 19 ms, and 20 ms. Keep in mind that because the timer is only 8 bits, there's an extra interrupt before the 18 ms one. The wheel state variables determine when the signals go high (if at all) at each of the points. So depending on the wheel state variables, the pulse width may be 1 ms, 2 ms, or no pulse. See **Figure 11** below for a pictorial representation of this description.

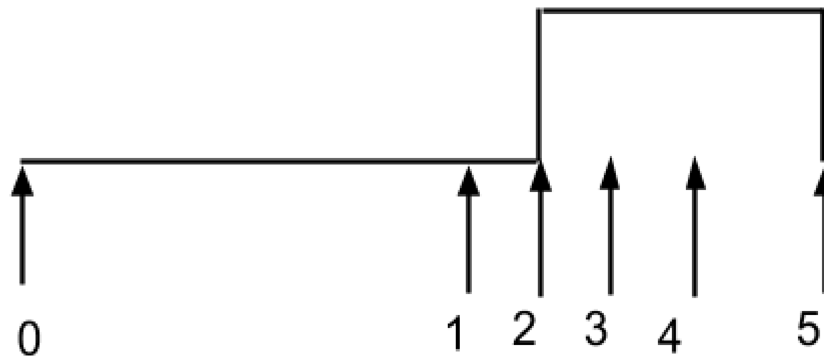


Figure 11: Pictorial description of where interrupts occur on the waveform

Each number in the picture corresponds to a state in **Figure 10** on the previous page. The wave goes high at state two in the picture, but it's also possible that the wave could go high at state three, four, or not at all depending on the wheel state variables.

Rack Motor Driver

The rack motor is simple from the software's perspective (the complexity is in the motor control circuit which will be described in the hardware section). The motor has a VCC line and a ground line. The motor needs to move up *and* down however, not just up. This accomplished by changing which wire is ground and which wire is vcc. So all the code needs to do is assign the correct values to the wires depending on whether MoveClawUp() or MoveClawDown() is called.

Claw PWM

The claw PWM works in almost the exact same way as the wheel control code. Please refer to **Figure 12** on the next page.

```
void FullyOpenClaw()
{
   OpenClaw();
   _delay_ms(2000);
   StopClawOutput();
}

void OpenClaw()
{
   timer1Behavior = OpenClawISRHandler;
   TCNT1 = 0;
   OCR1A = 31;
   TIMSK1 = (1 << OCIE1A);
}

void OpenClawISRHandler()
{
   if (openClawState == 0)
   {
      PORTB |= PWM_PIN;
      OCR1A = 31;
      TCNT1 = 0;
      openClawState = 1;
   }
   else if (openClawState == 1)
   {
      PORTB &= ~(PWM_PIN);
      TCNT1 = 0;
      OCR1A = 281;
      openClawState = 0;
   }
}

void StopClawOutput()
{
      TIMSK1 &= ~(1 << OCIE1A);
}
```

Figure 12: Open Claw Functions

When FullyOpenClaw() is called, OpenClaw() is called and the code is delayed. OpenClaw() starts the PWM, the delay is how long the PWM will occur (or how long the claw should "open" for), and finally the StopClawOutput() call disables the interrupt for the PWM and the claw stops opening. This code is somewhat less complex than the wheel code because it uses a 16 bit timer instead of an 8 bit one. There's also no pulse width variability.

The interrupt drives the claw pin high for 2 ms and then drives it low for 18 ms. This process then repeats. The CloseClawVersion is exactly the same except for the timer values which define how long the pulse is high (In order to close the claw the pulse is high for 1 ms and low for 19 ms).

Misc Information About Code

As a note, all the timer1 ISR does is call a function pointer called timer1Behavior(). This function pointer is frequently updated throughout the code to alter the behavior of the ISR for whatever purpose the timer is being used for. This is necessary because the Arduino Uno has only one 16 bit timer.

Also, there is a mention of a serial library in the code. This is merely for debugging purposes. The serial API allows information to be sent from the uno to a computer terminal over a serial cable.

Hardware

The high level hardware diagram for the system is shown in **Figure 13** below.

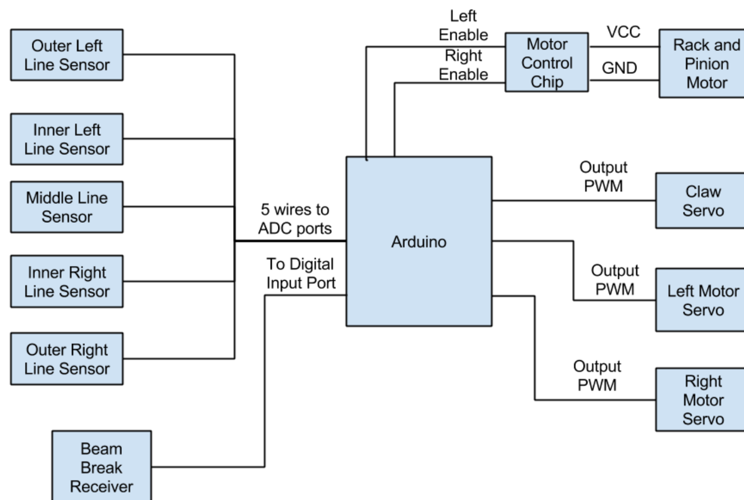


Figure 13: High Level Hardware Diagram for the system

The system works by taking line sensor data and beam break data and then controlling the various servos and motors based on the current inputs and previous inputs.

Line Sensors

The schematic for the line sensors is shown in **Figure 14** below.

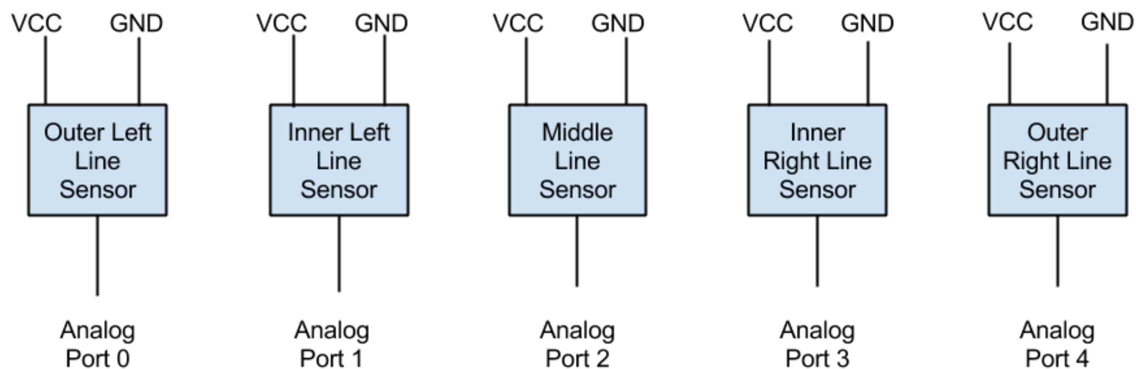


Figure 14: Line Sensor Array Schematic

The line sensors are powered by the 6 V supply rail (the power system will be described later in the report). Each line sensor then outputs an analog value to an analog in port on the arduino. These values are then handled by software.

Beam Break Sensor

The beam break sensor schematic is shown in **Figure 15** on the next page.

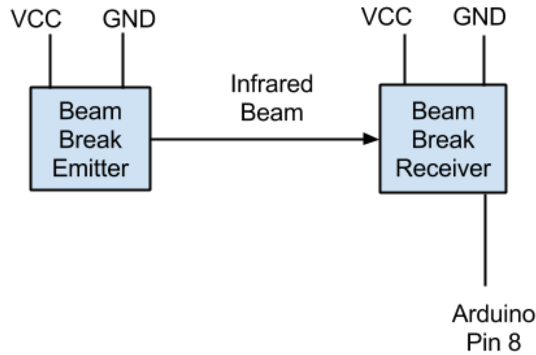


Figure 15: Beam break sensor schematic

There are two physical halves to the sensor. The beam break emitter creates an infrared beam and the beam break receiver looks for an infrared beam. If the beam is unbroken and the receiver recognizes an incoming beam, the output from the receiver will be high. If the beam is broken, the output from the receiver will go low.

Rack Motor

The schematic for the rack motor is shown in **Figure 16** below.

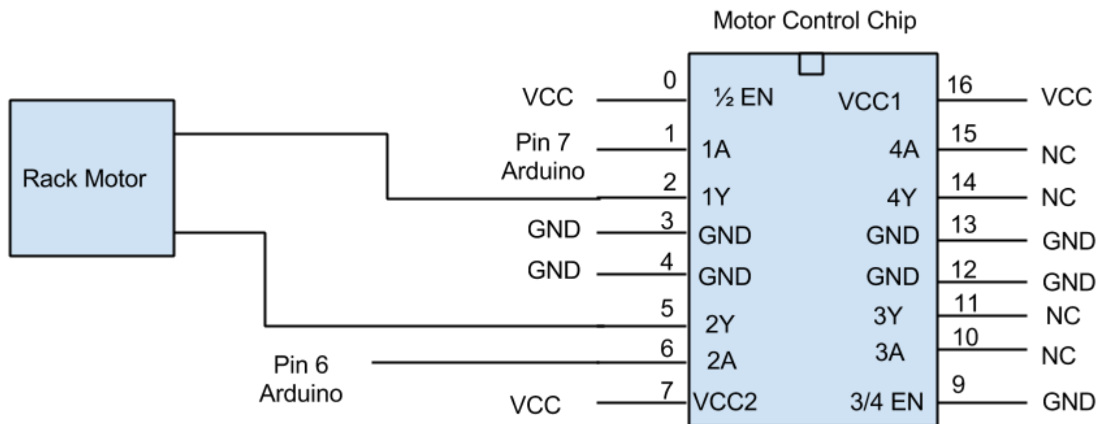


Figure 16: Rack Motor Control Schematic

The rack motor cannot be directly driven by the Arduino because it will draw more current than the arduino can supply. This problem is circumvented by using a SN754410NE motor control chip. A separate voltage supply is connected to the chip. When the enable pin is driven high (and the enable for the corresponding side of the chip is high), the supply will be passed through the corresponding output pin. In this way the motor can be controlled by the arduino without exceeded current draw capabilities of the board.

The chip is fairly simple to use. Voltage supplies are connected, the 1/2 , ¾, or both enables are driven high. The “A” pins in the schematic represent the enable pins (notice that the arduino pins are connected to these). When an enable goes high, the output (or “Y” pin) will go high as well, but with the voltage supply connected to the chip.

Servos

The schematic for the servos (full rotation) is shown in **Figure 17** below.

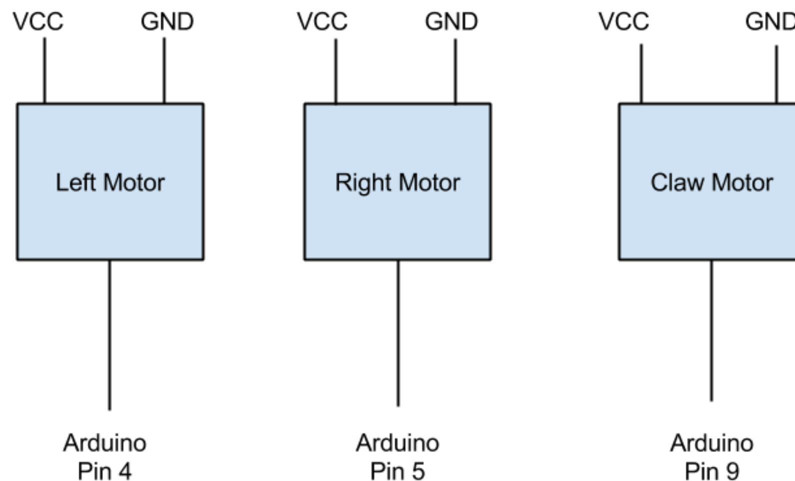


Figure 17: Servo Schematic

The servos are controlled through one wire. The duty cycle of the input wave determines which direction they rotate, as discussed in the software section.

Power Circuit

The schematic for how the system is powered is shown in **Figure 18** below.

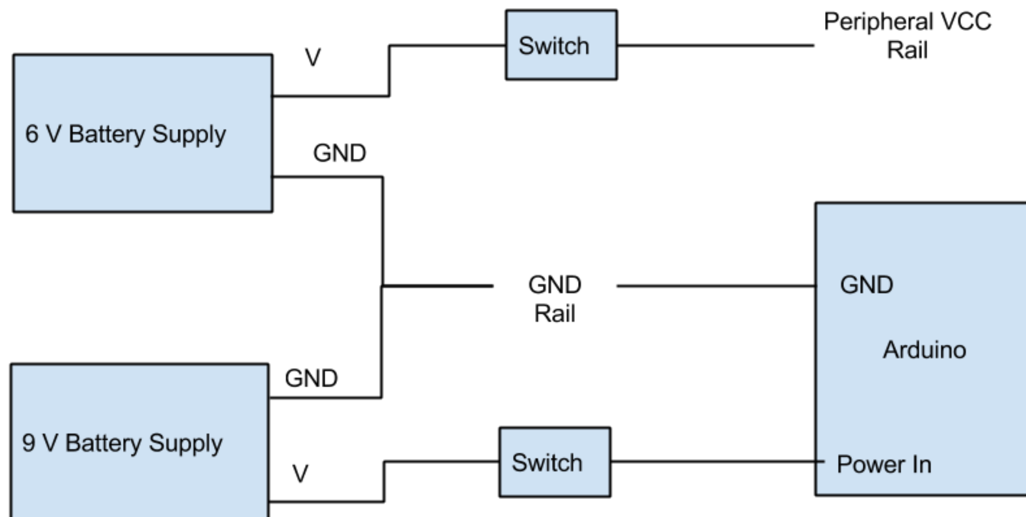


Figure 18: Power Control Circuit

The system requires two separate power supplies due to current output limitations on the Arduino. The system originally used only one power supply, but the arduino could not supply enough current for the motors. Therefore, a second pack was added from which peripherals are powered. Both supplies have a switch attached so that they can conveniently be turned on and off.

Mechanical Design

Claw

The mechanical drawing for the claw is shown in **Figure 19** on the next page.

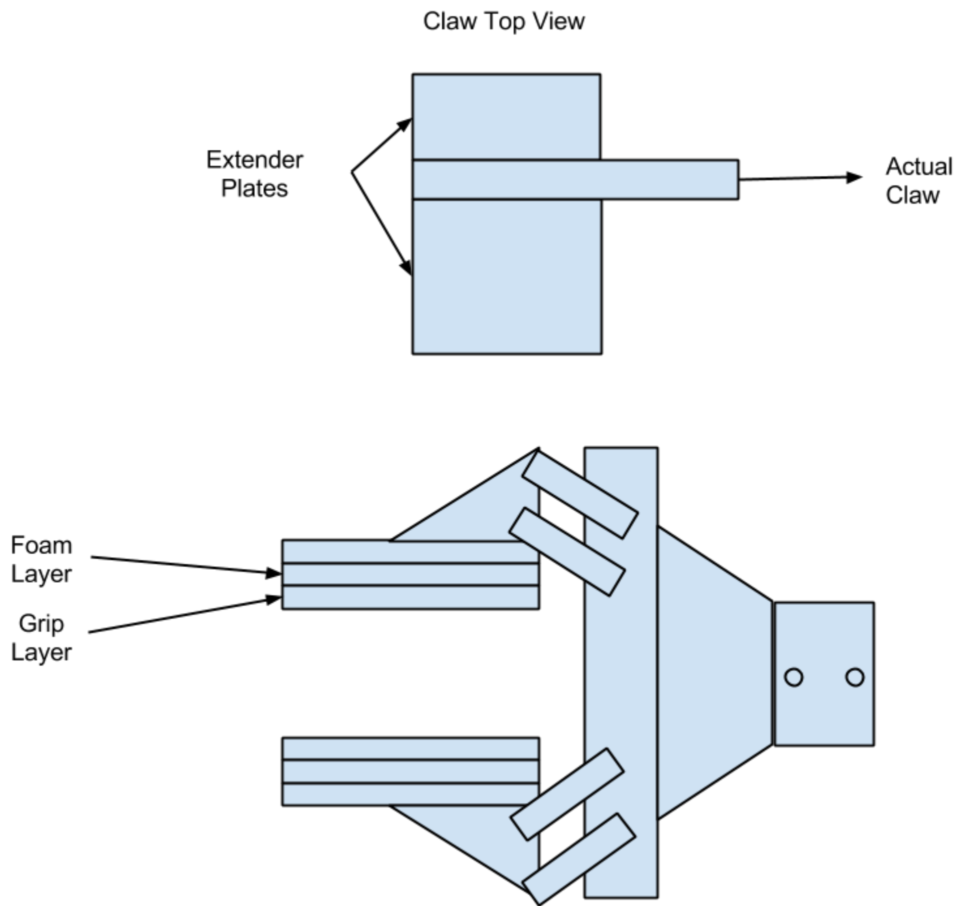
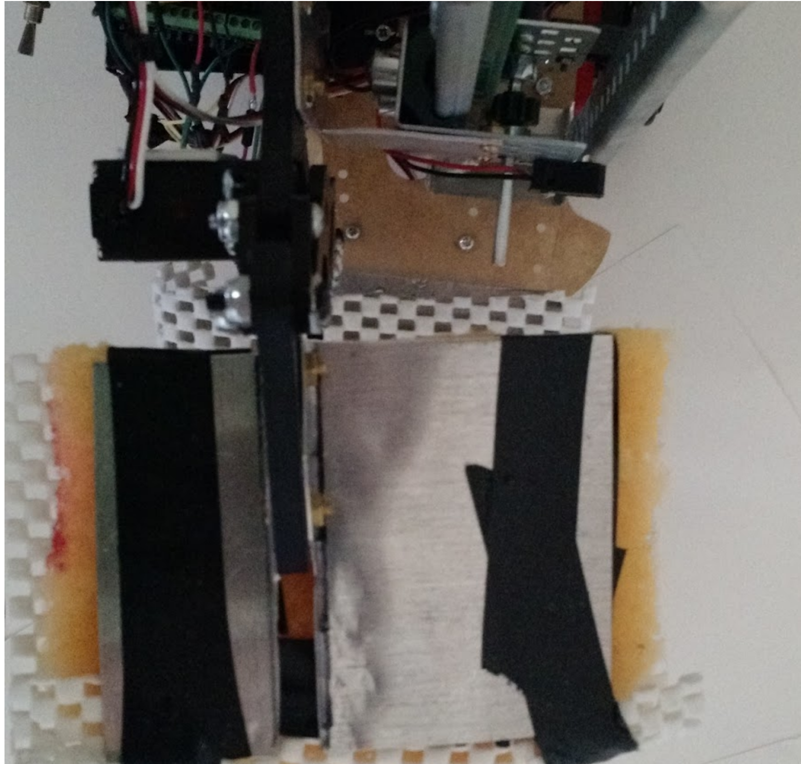


Figure 19: Claw Mechanical Drawing

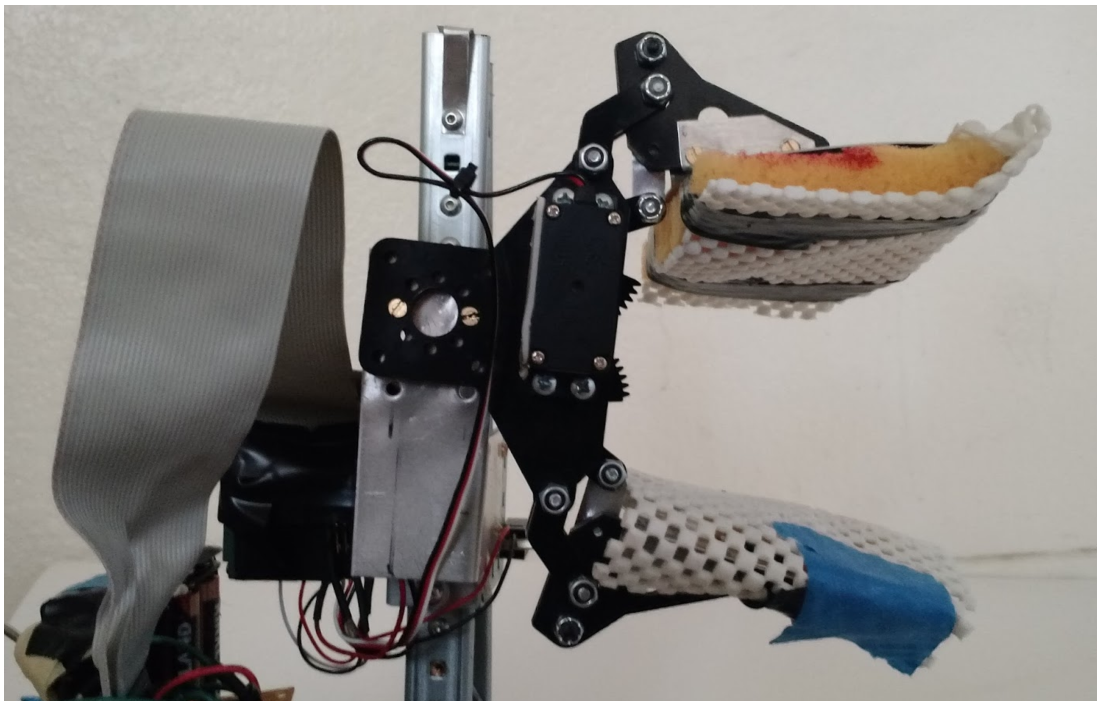
The gripping portion of the claw the team purchased was only about a fourth of an inch wide, which wasn't wide enough to properly grip all of the rings. Therefore, the team went to the machine shop and made custom extender plates out of sheet metal. These plates allowed the claw to grip the rings properly (even if the claw wasn't exactly aligned!).

Also, the claw had difficulty grasping the rings with just the metal as the contact point. Therefore, a layer of compressible foam was added with a layer of gripping material on top of that. Note that at certain weak grip points extra foam had to be added to compensate. This was done empirically during the testing phase.

See **Picture 1** and **Picture 2** below for the top and side view of the claw respectively.



Picture 1: Top View of the Claw



Picture 2: Side View of the Claw

Rack and Pinion

The mechanical drawing for the rack and pinion is shown in **Figure 20** below.

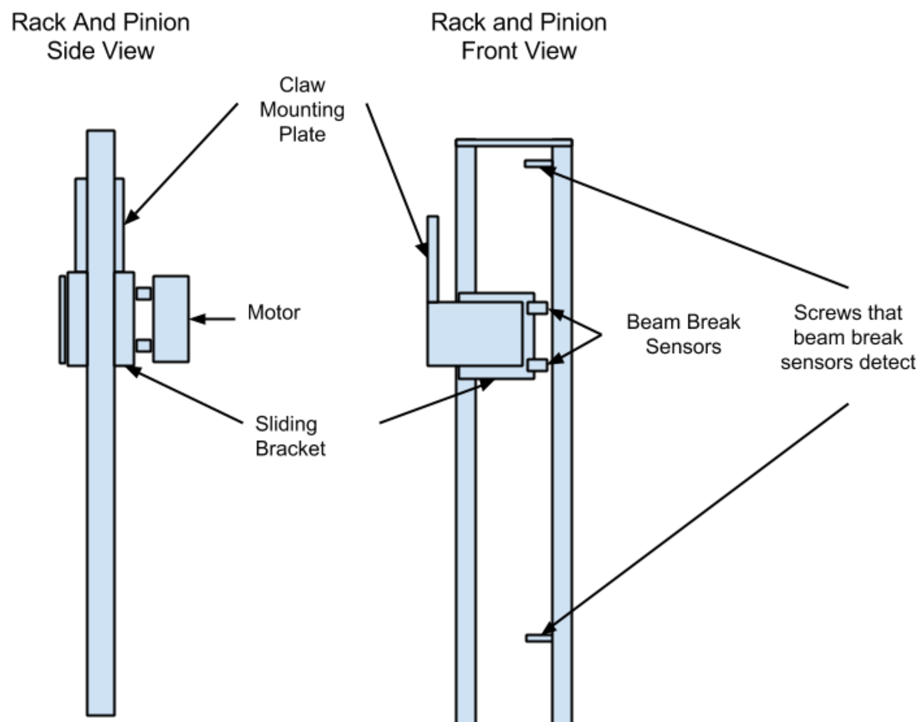


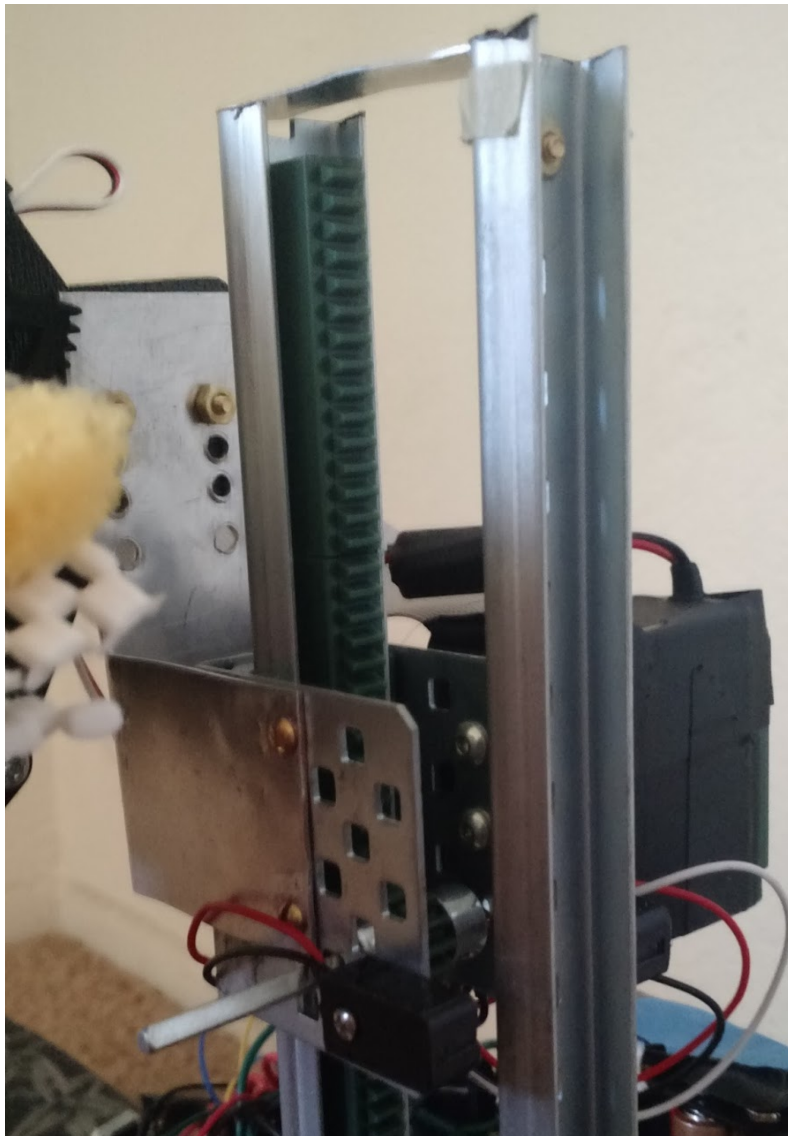
Figure 20: Rack and Pinion Mechanical Drawing

The rack and pinion moves the claw up and down. Note that the team decided that for simplicity we would only deposit rings on the highest vertical peg (hence only two "detection" screws for the beam break in **Figure 20** above. A motor is attached to a sliding bracket. The motor turns a gear in the sliding bracket that is attached to a toothed rail. Therefore, when the gear spins the the sliding bracket will move up or down. Beam break sensors are attached to the sliding bracket so that when the bracket slides past a screw the software can detect it.

Note the claw mounting plate attached to the sliding bracket. The bracket is necessary because of the initial maximum height of twelve inches at the start of the competition. The combined height of the robot base and the bars the sliding brackets move on is twelve inches. The claw mounting plate extends upwards, so that when the sliding bracket is at the top screw, the claw is actually higher than twelve inches and can therefore deposit rings on the top most vertical peg.

Most of the actual height requirements were determined empirically. Initial measurements were determined, however in practice they were somewhat off. When this was the case, the components were taken back to the machine shop and then altered accordingly.

See **Picture 3** below for an actual photo of the rack and pinion system.



Picture 3: The Rack and Pinion System

Front Bumper/Line Sensor Mount

The mechanical diagram for the bumper and line sensor mount is shown in **Figure 21** below (note that the whole front apparatus, not the base/wheels, was custom made).

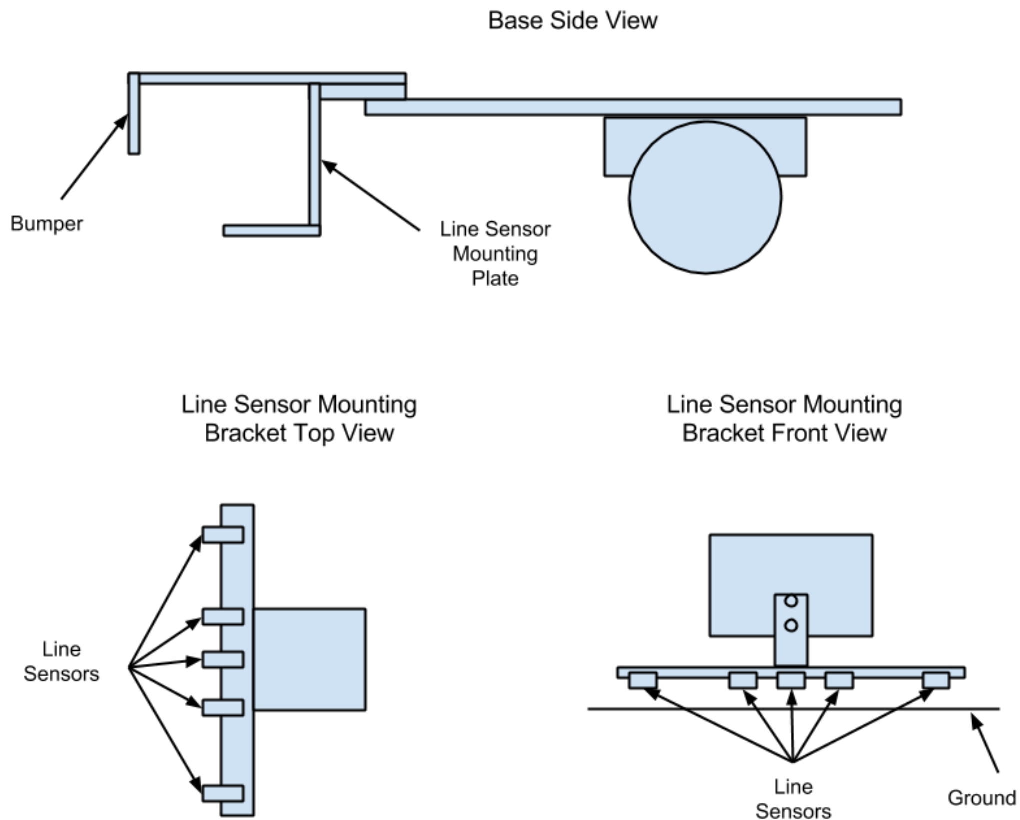
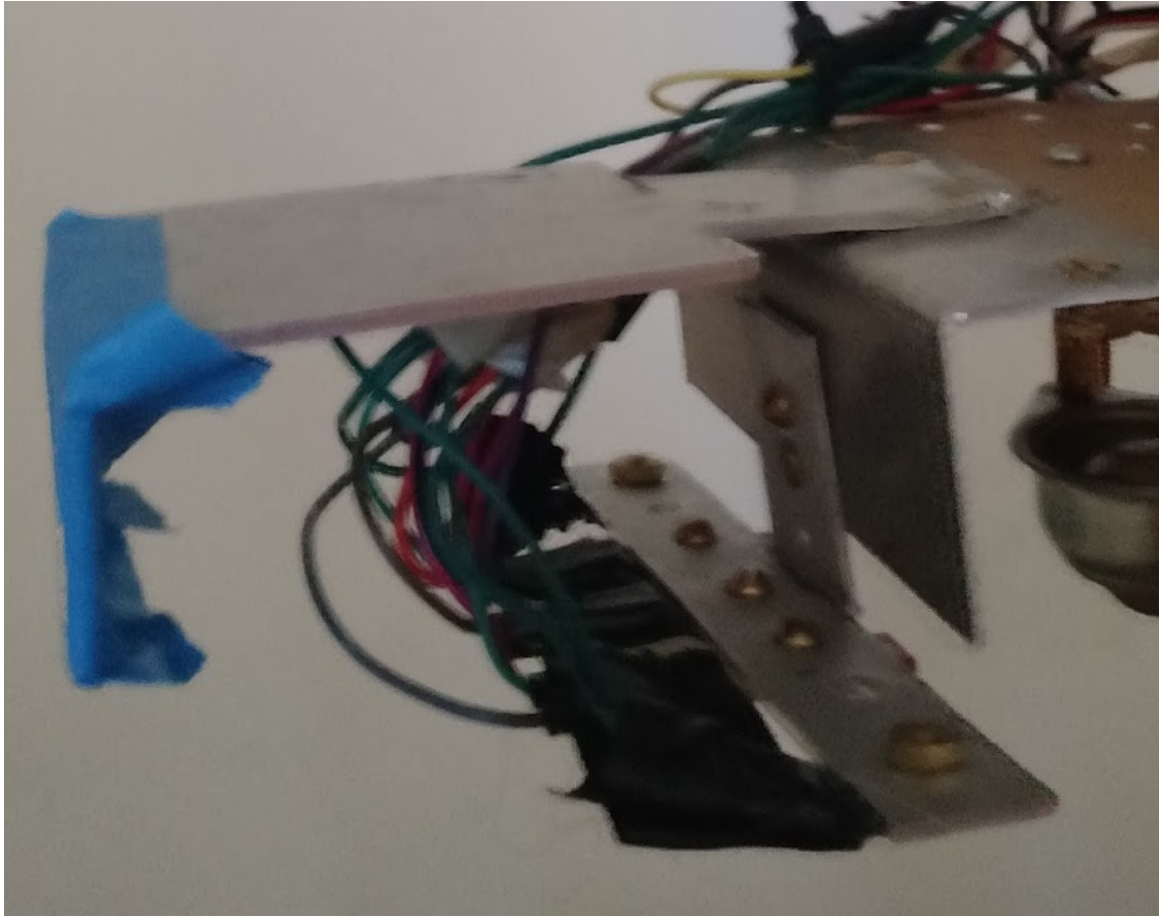


Figure 21: Mechanical Drawings for the front bumper

The front bumper is necessary so that when the robot drives forward, the back of the claw does not hit the PVC pipe. Otherwise, the robot would try and go forward too much and potentially tip over.

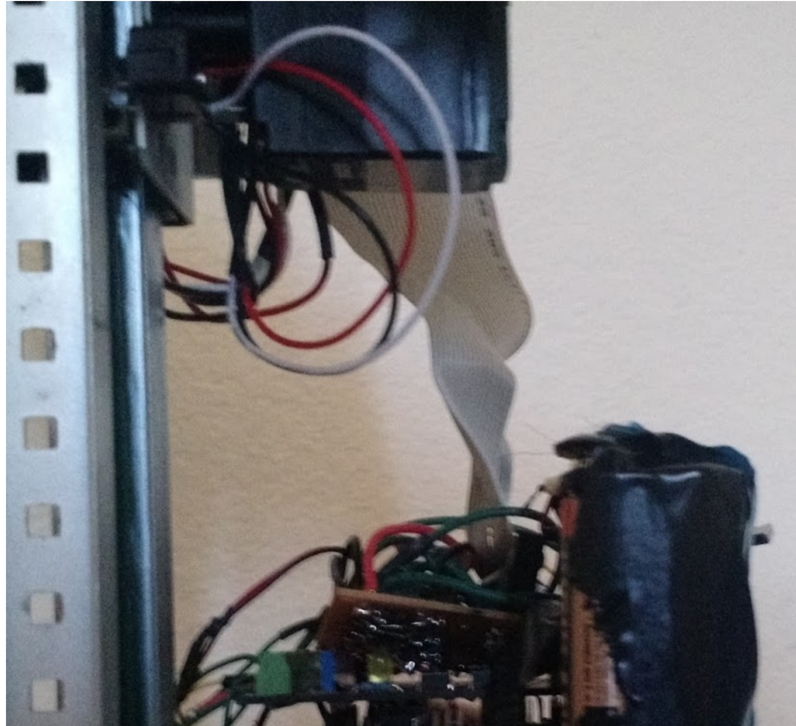
The line sensor mount keeps the line sensors at an appropriate height above the ground. They have a small range in which they work optimally, so the metal mount makes sure they are always at the correct height.

See **Picture 5** on the next page for an actual picture of the front of the robot.



Picture 5: The Front of the Robot

Also notice the base plate and wheels shown in **Figure 21** on the previous page. The base was purchased and the wheels were given to the team from our advisor. A metal plate was attached to the back of the base on risers to mount the electronics (the plate was insulated with electrical tape) and the arduino battery pack. The peripheral battery pack was mounted under this plate. See **Picture 6** on the next page.



Picture 6: Battery Pack Locations on the Robot

Budget and Bill of Materials

The project didn't have a defined budget, however the goal was to make the robot as cheap as possible. Unfortunately, the components from Vex Robotics (the rack/pinion system without the custom components we made) were quite expensive, driving the cost up significantly. Several failed design ideas were examined as well, again driving the cost up somewhat. Some components also had to be replaced after failures. The total cost for the project ended up being $338.02 dollars including all of the previously mentioned costs. See the Bill of Materials in **Table 1** on the next page.

Table 1: Bill of Materials

Component	Description	Quantit y	Price Per Unit	Total Cost
Analog Line Sensor	Senses lines	5	2.95	14.75
Robotic Claw	Claw that can grab	1	14.95	14.95
Chassis (Original)	Robotic Chassis	1	14.95	14.95
Full Rotation Servo	Servo to move claw	1	13.95	13.95
Digital Line Sensor	Digital Line Sensor	2	2.95	5.9
Linear Motion Kit	rack/pinion components	1	24.99	24.99
Advanced Gear Kit	rack/pinion components	1	19.99	19.99
Shaft collars	rack/pinion components	1	7.99	7.99
Drive Shafts	rack/pinion components	1	5.49	5.49
2-Wire Motor	rack/pinion components	1	14.99	14.99
Sheet Metal	Metal for custom components	1	19.99	19.99
Chassis (New)	New chassis after old one broke	1	14.99	14.99
Sponges	Sponges for claw padding	1	8	8
Grip Material	Grip material for claw	1	5	5
Misc Electrical Components	Switches, wire, solder, caps, etc	1	60	60
Batteries	batteries for peripherals	3	14.03	42.09
Course Test Materials	PVC, Board, Electrical Tape, etc	1	30	30
Screws/Nuts	Screws/Nuts/Washers	1	20	20
				0
			Total:	**338.02**

Lessons Learned

Mechanical Aspects of Parts and Their Integration

It would have been beneficial to pay attention to some of the structural properties of the parts and how they interacted. For example, the rack and pinion was placed on part of the base that allowed the base to oscillate when moving. After many of these oscillations occurring during the testing phase, the base eventually cracked and the team had to purchase a new one. Paying attention to issues such as this would have saved on the overall system cost.

Using appropriate wires and heat shrink

At the start of the project the system used purely solid core wires. Solid core wires are nice for plugging into an Arduino, but they break very easily. Early on, quite a few wires were breaking until we switched to stranded core. We also had many issues with solders breaking (even with stranded core). Once we started heat shrinking the solder joints, joints stopped breaking. Taking the time to make proper electrical connections is critical.

Pay more attention to power usage

During the project, we didn't pay a whole lot of attention to the systems power usage. In the end, Steve was eating through batteries at an alarming rate. It would have made more sense to have a battery bank that wouldn't die as quickly. Battery costs add up quickly, so it pays off to choose an appropriate power source.

Line sensors aren't great for detecting wheel revolutions

We thought it would be useful to be able to precisely calculate the distance wheels moved. Therefore, we covered the wheels in white paper and drew black lines at specific intervals. A digital line sensor hovered over the wheel to detect these lines, and the number of lines detected could be translated to distance since we knew how far the lines were apart. In practice, however, the line sensors had difficulty detecting the lines because the wheels were moving too fast. This is still a cool idea (suggested by our advisor), but a more appropriate detector would be needed.

Don’t use claw

We decided to use a claw at the start of the project (because it seemed like the most obvious choice), however it caused difficulties later on. For example, the claw position had to be extremely precise so that when opening the claw wouldn’t hit adjacent pegs. Also, it had difficulty grasping the rings (we had to do a significant amount of fine tuning to get it to work). Many other teams used a stick attached to a servo and that seemed to work much better.

Use better motors

The initial motors we had burnt out fairly quickly. On the day before the competition we had to swap them out for new ones (that used a different control scheme) and write the drivers for them. So it pays to understand how much a system’s motors can handle and whether the motors are appropriate for how the system uses them.

Bumper button

When the robot went forward to either grab or deposit rings, it would go forward for a fixed amount of time and then stop. This meant that the bumper would hit the wall and the robot would still be trying to go forward (which in hindsight may have contributed to the original motors burning out). Attaching a button to the front that detected when the bumper hit the wall would have avoided this problem.

Designed for simplicity and reliability, maybe too simple

We came up with several designs at the start and decided to pick the simplest one. Our robot grabbed the rings from the middle horizontal peg and deposited them on the top vertical peg. Nothing particularly fancy. We figured it would be smarter to pick something that would be more reliable and more practical to implement (since we did a lot of our work from scratch and not a kit). In retrospect though, it cost us somewhat. With the knowledge we have know, it would have been nice to be able to deposit rings on all of the vertical pegs and also to place a ring on the middle peg. That would have increased the number of points we received significantly.

Conclusion

Steve did significantly better than we expected. He came in fourth place, and almost came in 3rd (he was only behind three points that round, the match was 54-51!). Designing a robot is no easy task, primarily because it encompasses various fields such as mechanical engineering, electrical engineering, and computer engineering. Knowledge from all of these fields has to be pulled together to create a complete and functioning system. This is one of the reasons why robotics is so great, and why it was a pleasure to compete in Roborodentia.

Appendix A: Code

For the code in its entirety, please visit the following GitHub Repository:

https://github.com/jtdykstra/SeniorProject